Fundamentals of Multimedia Computing

Chapter 6: Brief Introduction to Human-Computer Interface Design

Authors:
Gerald Friedland
and
Ramesh Jain

Draft for Comments

Brief Introduction to Human-Computer Interface Design

While this is a book about multimedia computing and it would be easy to write another volume just on human-computer interface design, we felt this is absolutely the reason to at least include a chapter on it. In contrast to most other topics discussed in this book, human computer interaction has a largely subjective component. Like multimedia computing, human-computer-interface (or short HCI) research seeks to gather knowledge from cognitive psychology, especially in matters of perception. However, in addition to that, HCI research also has to take into account subjective aspects such as cultural bias and social behavior, and even issues such as political correctness. In other words: When dealing with HCI problems one is dealing first and foremost with problems regarding human beings. Informally, one could say the challenge is to make the computer be social.

So how can we make our multimedia application friendly to its human user? Of course, there is no easy answer to this question. In this chapter, we will therefore focus on only a few important aspects of HCI, which we think provide enough introduction for the reader to be able to start thinking in the mindset of an HCI person and continue studying the field further. We start with some general rules of user interface development and then continue with an introduction to human-based evaluation.

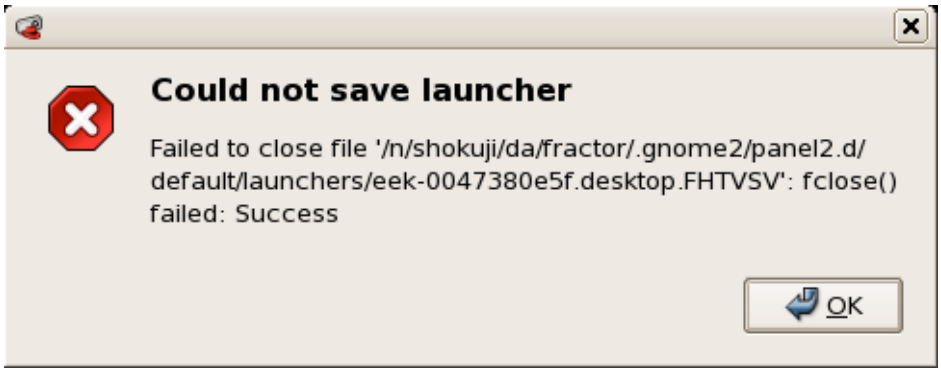

Figure 1. The developers of an application thinking inside-out is one of the most frequent cause for bad user interface design. The above dialog box shows a real-world example of a confusing dialog box that happened to present itself to the authors during the creation of this chapter.

Graphical User Interface Design

Most of today's applications, especially ones that support multimedia in any way, use a graphical user interface (GUI), i.e. an interface that is controlled through clicks, touch and/or gestures and allows for the display of arbitrary image and video data. Therefore knowing how to design GUI-based applications in a user friendly manner is an important skill for everybody working in multimedia computing. Unfortunately, with many factors that go into the behavior of a program and the perceptual requirements of the user, there is no unique path or definite set of guidelines to follow. Over many years and using research results and feedback from many users, however, some standards have evolved. These standards can be seen in many places today in desktop environments, smartphones, DVD players, and other devices. Very often though developers create a way to do things and then users adapt to it whether it is optimal or not, sometimes, users

would even argue about the way it is done in favor of one or the other paradigm: So is it better to have the menu bar inside the window of an application or is it better to have one menu bar that is always at the same place and changes with the application?[1] As we assume the reader knows, this is one fundamental difference between Apple's and Microsofts operating systems -- and it is hard to say one or the other is right or wrong.

Don't think inside out

Despite the discussion above there is one important rule that it is universally true for any machine or application interface: Never assume that the user knows what is happening inside the program (thinking inside out). The interface should assume no knowledge of any kind of he insides of the program. While this seems obvious, it is more than easy to accidentally require much more knowledge about a program than a user has because, after all, the creator of the program has intimate expert knowledge. Most user interface glitches are the result of a violation of this rule. Drawing the analogy to social interaction, this is similar to a person asking you to introduce yourself. Every person has many details he or she could tell about things happening everyday. Therefore, when we introduce ourselves, we start with social conventions, such as stating our name and then go on to a small selection of facts that are relevant in the context of the conversation. A program should follow the same guideline: It should present itself inside the user interface conventions of the particular device and then only expose those details that are required and expected by the user. If the program "introduces" itself with difficult technical details that are hard to understand, especially a first-time user will probably reduce his or her interaction to finding a way to end the program and think about a different solution to his or her problem. Figure 1 shows a bad real-world example of a developer thinking inside-out. It shows a dialog box of a modern, heavily-used application.

It is important to note that not all issues are as obvious as the one shown in Figure 1. In fact, most of them are not, however, Figure 1 shows you how severe the issue can become. Some of the results of thinking inside-out deserve their own category such as the next one.

Figure 2. A real-world example of technology rather than task-oriented development: Technology-wise it is easiest to think of any message as a dialog box. Task-wise this could result in disappointment and frustration for the user as an "OK" button in a normal dialog box usually suggest everything went fine. In this case a process failed, however, so the user should be presented a warning message.

[1] This particular example has been studied quite well though, see the references for details.

Don't develop technology-oriented but task-oriented

This is one rule that especially true for multimedia applications. A common example that demonstrates this is: Just because you have several hundred fonts to choose from it doesn't mean a document should use all of them. In fact, typical publications only use 1 to 3 fonts because most readers would not find it appealing to look at a mess of fonts. In multimedia applications one has the tendency to present the user with as many output, eg. different media, as one can. In reality choosing wisely is key here and often less is more. This will be discussed later in the chapter again. Most importantly, one has to focus on what the task is and what the user is capable once the technology fulfills all requirements. Figure 2 shows a more subtle example of this problem: A user is presented with a dialog box when a problem occurred. A warning message box should have been used. Technology-wise it is easiest to think of any message as a dialog box. Task-wise this could result in disappointment and frustration for the user as an "OK" button in a normal dialog box usually suggest everything went fine. In this case a process failed, however, so the user should be presented a warning message. In general, when designing the user interface to your application you should ask yourself the following questions:

- What are the (mental and physical) capabilities of the user, e.g. what are the technical terms understood by the user?
- What are the tasks that the program is helping the user to achieve and what information has to be presented and asked from the user for doing so?
- What are the upper and lower limits for resource requirements needed to accomplish the tasks, e.g. how much disk space is required and how much memory has to be allocated? A typical eror made here is to restrict array lengths arbitrarily and exposing the limitation to the user. Many of the Y2K bugs fall into this category, as developers underestimated how long their application would be used. In the multimedia community, screen resolutions and sampling rates have often been part of this problem.
- How can the application make the most important tasks easy and quick and the less frequent tasks at least possible? In other words: Find the right compromise between the ease-of-use and the complexity of the program.

Larry Wall, the creator of the PERL programming language is often quoted saying: "Common things should be easy, advanced things should be at least possible". Especially for multimedia applications this means to have the function in mind first and then the presentation. The most common issue for multimedia computing in this regard is probably a device that everybody has at home in abundance: The remote control.

Remote controls are inherently technology oriented, especially multi-device remote controls which can be used to command TVs, DVRs, and home theater systems alike. While some of the buttons, such as volume control, program up and down, and play and pause show standardized symbols, most other buttons look differently on every type of remote control, and most reader will agree that the function of some buttons remains a mystery forever. A task-oriented remote control, would only show buttons that are interesting to the user at a given point (e.g. when the DVR is used to playback a recording, the program buttons could be grayed-out). Also buttons that are rarely used should be hidden in an advanced setting. Of course, this is harder to realize with a physical remote control but it is possible in a display-based remote, e.g. on a cell-phone. Therefore, we dedicated an exercise to it.

Don't be creative when you are not supposed to

Think about the following social situation: You meet with your friends to play soccer. While have been doing this together for a long time, this time you decide to change the rules. What makes it worse, you don't even tell them, but you just play by your new rules. Small children know that such a behavior will most likely lead to confusion, frustration, and conflict. The same applies to user interface design. When the device or operating system manufacturer sets a standard for certain GUI elements to behave a certain way, your program should not change the behavior. So the first step is to understand what the rules for a certain GUI element (such as a modal dialog box) are and the second step is to never ever use the element in a different way than it is supposed to. Like the soccer example, doing so will lead to confusion and frustration of the user and also potentially lead to conflict with other applications that share GUI elements. Some manufacturers also enforce GUI standards and will likely cause you to redo your interface. While this rule is true for any application, multimedia applications are the most prone to tempt you to abuse predefined UI components. The reason for this is that conventional operating system interfaces are originally created for text and mouse input and output. Multimedia applications, however, might deal with various inputs and might requires content-based selection and editing operations. Figure 3 shows an example of a system that uses a chalkboard metaphor instead of a desktop metaphor. As of the writing of this book, most of these are not yet standardized. The recommendation here is to stay as close as possible to the interface standard and only takes as much freedom as is required. Finding the right and intuitive solution might require user interface studies, as explained later in this chapter. A firm rule here is that the user should never have to perform unnatural or unintuitive actions.

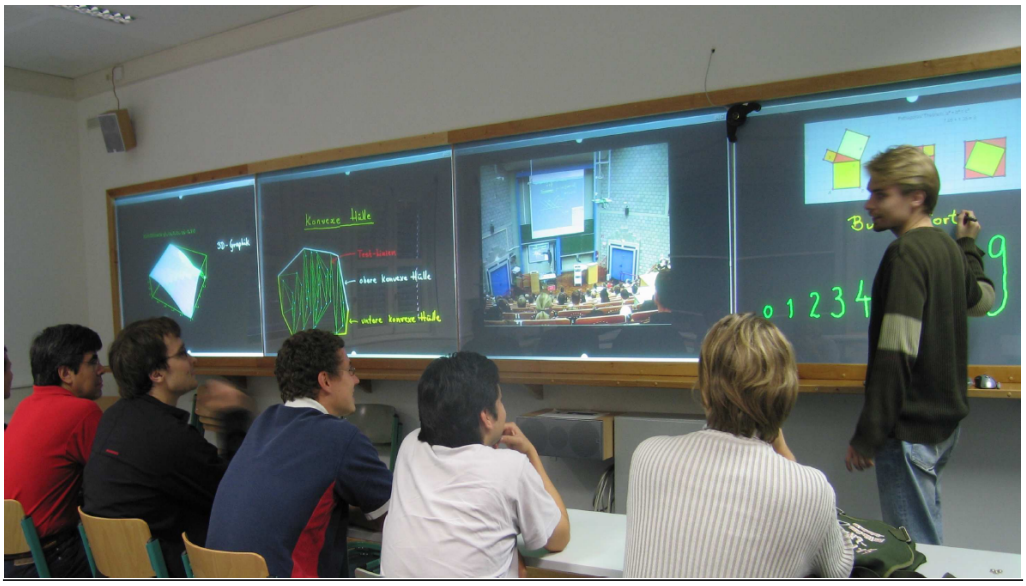

Figure 3. The electronic chalkboard system E-Chalk is an example of a multimedia system that would not work with the desktop metaphor.

Foster the learning process of the user

Your goal as an application developer should be to bind the user to your program for a long time. This is best achieved when the user incorporates your application more and more into daily live

and thinks of it increasingly often when a new problem comes up. On the first use, the program might just have solved some minor problem but when the user's increasing understanding of the complexity of your program is rewarded with the application solving more sophisticated problems, it is very likely that the application becomes an important tool in a user's life.

Let's again start with a bad real-world example: Figure 4 shows how to make sure even an experienced user has to think twice about what to do. To foster the learning process, your application interface should follow one concrete philosophy (see example later in the chapter). For example, many operating systems follow the Desktop metaphor, therefore using terms like "folder", "file", and/or "trash bin". Sticking to the metaphor or philosophy is very important, inconsistencies or contradictions should be avoided. Most importantly, the environment should be made riskless along the philosphy. For example: Like in real life, things put in the trash bin, can also taken out of it when they haven't been in the bin for too long. Undo and redo functionality make using computers for typewriting even less risky than using a physical typewriter. Riskless environments promote learning because making a mistake is not penalized heavily encouraging exploration. Another examples is intelligent default values: They help exploration because the user can see what happens before understanding what the values mean.

Another property of a program that promotes learning is transparency. Of course, the user does not always have to know what is going on -- this would contradict the first paradigm of not thinking inside out. However, the user should always know what is expected of him and what he can expect from the computer. So if a certain action triggers an LED to light up, it should always do that. Programs should also not start tasks on their own without educating the user about it. Numerous system tools, such as virus scanners, have violated that rule in the past, by starting background or update processes in the background with the result of user becoming frustrated. A typical multimedia example here is fast-forwarding or rewinding of a video using a typical Internet player interface: Often it is not clear how long the process will take as it depends on the encoding of the video and the bandwidth of the connection between video server and player client. So many users don't even want to touch the controls anymore once a video has started playing. This issue is called responsiveness and is dedicated it's own section.

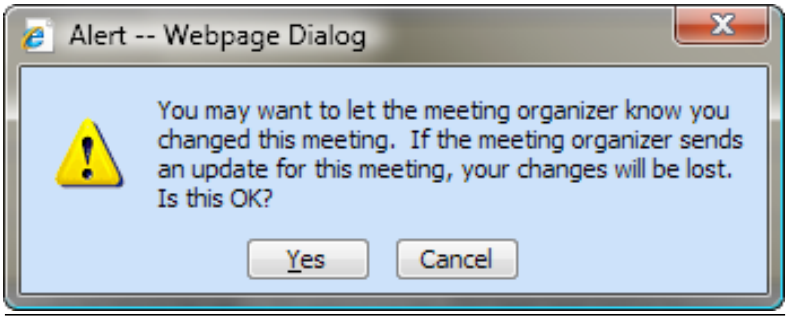

Figure 4. Fostering the learning process the wrong way: Even an experienced user will have to think twice to know what button to press.

Responsiveness

Responsiveness refers to the ability of a program to complete a task within a given time. The paradigm of transparency requires to educate a user about tasks that will take longer than an

expected time span. For example, when the computer is not responsive due to a compute or I/O-intensive operation, a busy cursor should be displayed, otherwise the user will think something is wrong. If the operation takes longer than a while the operation should be cancelable and a progress indicator should indicate the estimated time until completion of the operation.

Responsiveness has been studied, especially in connection with multimedia computing (see references XXX). The general rule is that when a program is not able to respond to user interaction for more than about one to three seconds (depending on the general responsiveness of the device) the user should see an indicator for this being normal, e.g. an hourglass cursor. Other rules of thumb include the following: If an operation occupies the machine for more than about ten seconds, the application should show a progress bar. Operations of 20 seconds or longer or when the duration is hard to determine should show a progress indicator and should also be easily cancelable. Operations that take several minutes, e.g. the conversion of videos from one format to another, should inform the user of this and prompt confirmation.

Sometimes it can be tricky to assess the time duration of an operation because the computation time to perform a task of course varies from computer to computer, depends on the data being processed and may change with available bandwidth. One solution to the problem is sometimes to perform a small subset of the task (e.g. convert only a couple seconds of a video) and then estimate how long the completion of the entire task will take. It is usually acceptable to slightly overestimate. Another solution might be to perform the task in the background (e.g. on a secondary CPU core) and have the computer not be unresponsive to user interaction. Rather than disabling all the interaction, only interaction that depends on the task being completed is disabled. For example, further operation on a video being converted might be prevented but loading a second video might still be enabled. So the user may choose to perform a different task instead of waiting. Of course, it must be transparent to the user what the computer is occupied with and what interaction is still possible. A third solution, that works particularly well with many multimedia applications is to visualize the progress in addition to showing a progress bar. For example, when transforming a video, the program could show the frame currently being processed. While this does not cut down on processing time, in fact it will most likely increase processing time, it might cur down on perceived processing time as the user is engaged in the visualization of the content. As explained above, human interface design is not about objective numbers, it is about subjective satisfaction. The next sections will discuss this in more detail.

Test your program with real users

The above stated principles and rules are neither comprehensive nor do they guarantee success of an application. Furthermore, let's assume you think your program presents itself in a learning-promoting way and hides exactly the right amount of details. How can you be sure? After all, you might be thinking inside-out. Therefore, the most important rule is to always test your application with real users. As explained above, human computer interaction research is evaluation based. The following parts of this chapter will introduce the basics of human-based evaluation.

Evaluation of Software through Human Subjects

Dix et al. (see references) formulated a simple equation for the measurement of the usability of a program:

Usability = Effectiveness + Efficiency + Satisfaction

Of course, quantifying the parameters of the equation is not only complicated but also subjective. Also, for some applications, different parameters might have different importance. In a computer game, for example, satisfaction probably equals effectiveness. Depending on the game, efficiency might or might not matter. So how does one measure the usability of an application?

Again, there is no silver bullet method to measure the usability of an application. An often used method in industry is to equal user satisfaction with profit. For example, one of two webpage designs is shown to new visitors at random and then the company appearance is further optimized in the direction of the page that results in most sales. The method is called "split test". The main drawback of the method is that sales numbers might correlate with many other factors and decreasing sales numbers might not be caused by bad webpage design. Two scientific methods are the video surveillance tests and questionnaires. Of the two, video surveillance tests are probably the most effective.

Video Surveillance Test

For a video surveillance test, candidates are chosen that represent the typical users of a particular application. Inside the target group it is best to select persons to represent a variety in gender, age, ethical, social, and educational background. The candidates are then given a set of tasks to solve with the program. Sometimes it can be effective to also set a time limit. With the consent of the participants, video equipment records audio and video of the users behavior together with the screen of the running application. It is best to not at all interfere with the test participants and to isolate the test subjects so that they cannot interfere with each other before, during, and after the test. Of course the tasks should be chosen wisely and probably in order of increasing difficulty. The tasks should be formulated general enough so that a user does not only have follow instructions but has to figure out the steps to achieve the task on his or her own. It may sometimes make sense to ask for tasks that are not achievable. In any ways, to avoid frustration, clear instructions must be given what to do in the case that a user does not find a solution, eg. that he is allowed to skip a task. One of the most important suggestions with any kind of user study is that the higher the number of participants ("higher n") the better. More participants will not only allow for a significant result but also help to find errors both in the application and in the test setup. Unfortunately, still too many user studies are often performed using "*n=20* students from my department". Granted, a video surveillance test with a high number of users is expensive and time consuming.

Questionnaires

To reach a greater audience, eg. over the Internet, questionnaires can be used. The most important problem with questionnaires is that only problems can be caught that the maker of the questionnaire was aware of. While adding free form space to a questionnaire is possible, one should not expect the most extensive feedback from it -- after all, testers are not software developers!

Here is a real-world example to illustrate the problem: A well-known beverage manufacturer created a new vending machine. They used a video surveillance test to find out about users'

reactions to interacting with the machine. After watching all the videos and discussing between management and developers, they found that everything seemed to be in order, except the user experience can be improved by an option to not only pay with coins but also with bills. So it was added to the machine. Then, to quickly prove that the change was successful, another user study was performed. This time, many more people were asked to participate and the only thing they had to do was fill out a questionnaire rating their experience using the machine on a scale from 1 to 10. As control, the old vending machine was tested with the same questionnaire. Now, most questionnaires for the new machine had a worse score than the control test using the old machine. Development and management were devastated: How could adding a feature that people wanted to a vending machine that was already quite good make the experience so much worse? The solution is this: When the vending machine only accepted coins, participants had been told in advance to bring coins. So the vending went smooth and comments in the videos were mostly about details. Now, that the vending machine also accepted bills, the participants were not told anything about payment modalities. As a result, most participants ended up not having coins and tried the bill slot. This would have been no issue, except the bill slot happened to not have worked very well because the bills could only be inserted in one particular way which the developers had already trained themselves to. As a result, the machine did not work at all for many subjects since the bill was rejected several times, resulting in many participants just giving up. So while the rest of the vending machine stayed the same, adding a feature made the user experience of the vending machine much worse and finding out about it was hard from the survey questions as developers expected them to be mostly positive.

Still, when done right, questionnaires can provide valuable knowledge. It is usually best to try a video surveillance test using a small set of participants first, address the issues raised as a result of the test in the application, and then create a questionnaire that targets the improvements in the application along with other issues raised in the video surveillance test. Thorough reviewing of the questionnaire is very important. Small choices in wording can make a large difference. Certainly, no wording should offend anybody or be hard to understand. A good questionnaire has questions with slightly overlapping semantics so that tendencies and contradictions can be extracted from the responses. Contradictions in response usually indicate somebody did not care one way or the other. Another important choice is to whether the questionnaire contains yes/no responses, rating on a scale (e.g. 1 to 10), multiple choice answers, or requires free-form responses. Free-form responses are the most accurate, yet take the most time to fill out. Unless appropriate incentive is provided for filling out the questionnaire, it is hard to have somebody paying attention to it for more than 5 to 10 minutes. Of course, the questions in the end might then be skipped or if multiple choice questions be answered randomly. It is therefore suggested to put the most important questions in the beginning of the questionnaire and have the questions in the end not be multiple choice. Yet again, with any multiple choice questionnaire there is no scientific way to tell how much thoroughness a participant puts into the responses -- especially when the crosses are not distributed arbitrarily. One way to add motivation to a questionnaire is to make the participant feel that he is contributing and his or her voice will make a difference. A large number of well-filled-out questionnaires will lead to significant results that are reportable in scientific publications and to an improvement of the application.

Significance

Especially when surveys are performed for scientific publications the question often arrises whether the outcome of the responses to a question on a questionnaire is statistically significant, i.e. when performed for a second, third, and further time, would the questionnaire lead to the same results? In other words: Is the outcome of this questionnaire random or a valid result. Fortunately, statistics can help here: Significance testing allows the survey analyst to assess evidence in favor of some claim about the participants from which the sample has been drawn.

Every significance test begins with a so-called null hypothesis. The null hypothesis represents an assumption that has been put forward, either because it is believed to be true or because it is to be used as a basis for argument. For example, in a questionnaire about the next version of an application, the null hypothesis might be that the new version is no better on average than the past version. The so-called alternative hypothesis is a statement of what a statistical hypothesis test is set up to establish. For example, in the evaluation about the new version of the program, the alternative hypothesis might be that the new version has a different target group, on average, compared to that of the past version.

The final conclusion, once the significance test has been carried out, is always given in terms of the null hypothesis. One either rejects the null hypothesis in favor of the alternative hypothesis or one does not reject the null hypothesis. If one concludes to not reject the null hypothesis, this does not necessarily mean that the null hypothesis is true, it only suggests that there is not sufficient evidence against it in favor of the alternative hypothesis. Rejecting the null hypothesis suggests that the alternative hypothesis may be true.

The significance is measured as number in the range between 0 and 1 and is usually denoted with α. If a significance test gives a probability value lower than the α-level, the null hypothesis is rejected. Such results are often informally referred to as statistically significant. The lower the significance level, the stronger the evidence required.

Here is an example: A male subject is tested for clairvoyance. He is shown the reverse of a randomly chosen play card 25 times and asked which suit it belongs to. The number of correct answers, is modeled by the random variable *X*. As the test goes on to find evidence of his clairvoyance, the null hypothesis is that the person is not clairvoyant. The alternative hypothesis is, of course: the person is (more or less) clairvoyant.

If the null hypothesis is valid, the only thing the test person can do is guess. For every card, the probability of guessing correctly is 1/4. If the alternative is valid, the test subject will predict the suit correctly with probability greater than 1/4. We will call the probability of guessing correctly *p*. The hypotheses, then, are: null hypothesis: $H_0 : p = \frac{1}{4}$ (just guessing) and alternative hypothesis $:H_1 : p > \frac{1}{4}$ (true clairvoyant).

Of course, if the test subject correctly predicts all 25 cards, one can consider him clairvoyant, and reject the null hypothesis. With fewer hits, let's say 24 or 23, the probability of the subject being (a pretty good) clairvoyant is still pretty high until one hits the lower numbers where p<=1/4.

Other tests of significance exist that assume statistical distributions differently from the uniform distribution, such as the Z-Test (see references) which assumes Gaussian distribution. All in all, questioning the significance of an outcome of a test becomes more and more an issue, the more test subjects are involved and the clearer the outcome of the response is. In the end though, mathematical significance testing is not a guarantee for actual results.

Consider the following real-world example: To test a new audio codec, the developers compressed several audio files using the different codecs and let users in the Internet listen to the codecs and rate them according to their perceived quality. To their big surprise, the codec with the highest bandwidth got the worse results, which was obviously better then the worse codec whenever it was presented to individuals outside the test. The test was definitely repeatable and statistically significant. What had happened? The developers had used different audio recordings. However, the one using the highest-bandwidth codec was a speech recording from a non-native speaker which some of the participants had a hard time understanding. This lead the test subjects to give the codec a bad score since they had only been asked to rate "the perceived quality". So the mistake here was to not eliminate all factors. A short interview after the test might have helped eliminating this oversight in the first place.

Again, this example shows, there is no silver bullet and human-computer interaction is a hard soft topic (hard in the sense of difficult and soft in the sense of not strictly logical). The reader is encouraged to delve into the literature for more information but, most importantly, to go ahead and try it out!

Literature

- Jeff Johnson: GUI Bloopers: *Don'ts and Do's for Software Developers and Web Designers* (Interactive Technologies), Morgan Kaufman, 1st Edition, March 31st, 2000
- Jakob Nielsen: *Usability Engineering*. Academic Press, Boston 1993
- Donald A. Norman: *The Psychology of Everyday Things*. Basic Books, New York 1988
- Jef Raskin: *The humane interface. New directions for designing interactive systems*. Addison-Wesley, Boston 2000.
- Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale (2003): *Human–Computer Interaction*. 3rd Edition. Prentice Hall, 2003.
- Helen Sharp, Yvonne Rogers & Jenny Preece: *Interaction Design: Beyond Human–Computer Interaction, 2nd ed. John Wiley & Sons Ltd., 2007*
- Matt Jones and Gary Marsden (2006). *Mobile Interaction Design*, John Wiley and Sons Ltd.

Web Links

http://www.gui-bloopers.com/

Exercises

1. List examples of thinking inside out that do not pertain to UI development.
2. What are the major advantages and disadvantages of command-line interfaces?
3. Give one example where the desktop metaphor fails and explain why.

4. Create a program to measure the time it takes to find a random specific point on the screen and click on it with the mouse. Then create a program that asks to press for a certain key on a random point on the screen and measure the response time. Compare the two and discuss.
5. Design a task-oriented remote control that can control a TV and a DVR.
6. Explain the notion: “Responsiveness = Perceived Performance” and give more examples of techniques that help increase perceived performance.
7. Find 3 GUI bloopers in the Internet that fall under the category “don’t be creative when you are not supposed to”.
8. Choose an office program (e.g. OpenOffice writer) and create three tasks that a user should solve using the program. Choose three test subjects from your friends and family and watch them while they are performing the task. If you can, you should use a camera to make sure not to interact with them.
9. Repeat 7 but this time create a questionnaire. What is different?
10. If you are restricted in the number of subjects to be participants in a user study (e.g. n=20) -- what is a good strategy to still obtain significant results?
11. Take the clairvoyant example from the text: How many cards have to be tested minimally for the test to be equivalent to a full 25 card test?